International Journal of Electrical and Computer Engineering (IJECE)
Vol. 5, No. 3, June 2015, pp. 586~598
ISSN: 2088-8708


# Application-level Communication Services for Development of Social Networking Systems

**Yunjin Lee*, Mingyu Lim**, Yangchan Moon****
* Division of Digital Media, Ajou University, Korea
** Departement of Internet & Multimedia Engineering, Konkuk University, Korea

**Article Info**

***Article history:***

Received Jan 16, 2015
Revised Apr 21, 2015
Accepted May 6, 2015

***Keyword:***

Client-server system
Communication middleware
Content transmission
Social networking service
User membership management

**ABSTRACT**

In this paper, we present our communication middleware (CM), which is designed to reduce the effort of developing common communication functionalities for social networking services (SNSs) in the client-server model. SNS developers can apply the application-level communication services of CM both to an SNS server and to client applications simply by calling application programming interfaces (APIs) and configuring various options related to communication services. CM was developed to enable SNS developers to easily build fundamental services such as transmission of a user-defined event, user membership and authentication management, friend management, content upload and download with different numbers of attachments, chat management, and direct file transfer. All of the communication services also provide options that a developer can customize according to his or her SNS requirements.



***Corresponding Author:***

Mingyu Lim,
Departement of Internet & Multimedia Engineering,
Konkuk University,
120 Neungdong-ro, Gwangjin-gu, Seoul 143-701, Korea.
Email: mlim@konkuk.ac.kr

## 1. INTRODUCTION

As wired and wireless networks have proliferated and become prevalent in our daily life, the prevailing network infrastructure enables users to easily create, process, and share social content anywhere, at any time, using various internet-enabled devices such as desktop PCs, laptop PCs, tablet computers, and smartphones. Among the popular services in such an environment are social networking services (SNSs), and the number of SNSs and their users are growing. Developers of existing SNSs have focused primarily on implementing various content services, but they have overlooked the inefficiency of development caused by redundantly providing common communication services. Although different SNSs have been developed separately, they have a number of similar communication-related functions such as user membership, authentication, event notification, sharing of text messages optionally having file attachments, and direct content transmission.

In this paper, we propose our communication middleware (CM), which aims to support the development of social networking services by reducing the effort required to develop communication services. As an application-level communication framework, CM provides SNS developers with simple application programming interfaces (APIs) and configuration options related to communication services such as arbitrary event composition and transmission, user membership and authentication, friend management, content sharing, chat management, and direct file transfer. Each service also provides options via CM configurations or API call parameters to customize the services for a variety of possible application requirements.

The remainder of this paper is organized as follows. In Section 2, we survey existing middleware systems for SNSs and compare them with our approach. In Section 3, we briefly introduce CM and its architecture. In Section 4, we describe the details of how CM options can be configured in SNS server and client applications. After we present how a developer can integrate CM into server and client applications in Section 5, we describe in detail how applications participate in a CM network in Section 6. In Section 7, we present the CM support functions for communication services, and with Section 8, we conclude the paper.

## 2. RELATED WORK

There have been some past studies on middleware services for SNSs. Brooker et al. [1] presented a middleware platform for developing social networking applications specifically for smartphone devices. With their middleware, not only can a smartphone request a service but it can also host a service with the help of surrogate clouds. One of the features for enhancing performance is an adaptive heartbeat mechanism that dynamically controls the frequency of heartbeat messages between a smartphone and a surrogate according to the service context. MobiClique [2] is mobile social networking middleware that directly disseminates content among opportunistically connected devices in ad hoc social networks. Two distinguishing features of MobiClique are that it does not depend on a centralized server and that it takes advantage of the social network overlay to disseminate content in a peer-to-peer manner. MobiSoC [3] is a middleware system that enables the development of mobile social computing applications and provides a common platform for capturing, managing, and sharing various social states of physical communities. To augment the social state, MobiSoC incorporates discovery algorithms for finding previously unknown geo-social patterns. Mokhtar et al. [4] proposed a middleware service for pervasive social networking environments. With this middleware service, a user can easily find other users who are socially or physically associated and can share common interests with them; this research focused on social interaction among users. Karki et al. [5] introduced a social networking approach for the mobile environment using PeerHood middleware. PeerHood [6] is a network management middleware module that provides a communication environment for mobile devices to communicate with each other directly, without any centralized server. The supported functionalities include device and service discovery, service sharing, connection establishment, data transmission, active monitoring of a device, and seamless connectivity.

In summary, existing SNS middleware approaches focus mainly on supporting their specific end-user services such as location services and searches for friends. However, they overlook common communication services, even though many SNS applications have duplicate and similar functionalities.

## 3. COMMUNICATION MIDDLEWARE (CM)

Our communication middleware (CM) is designed to support both synchronous and asynchronous interactions of users, especially within SNS applications. The main role of CM in an SNS is to support an easy and efficient way of developing an SNS application with high content accessibility. As middleware that is located logically below the application layer, CM provides a developer with APIs for various communication services such as communication architecture, user management, and event transmission.

We classify CM's internal classes into three modules: a controller module, a model module, and a normal module. The controller module contains classes that control and update internal values. The model module consists of classes that maintain the global information of CM. The normal module consists of the remaining classes, those that belong to neither the controller module nor the model module. Figure 1 shows the main classes in each module. The role of each class is described in detail in [7].

To provide platform independence, CM was developed with Java using the Eclipse Integrated Development Environment (IDE); thus, when compiled into Java bytecode, CM can run on any Java virtual machine (JVM) on a variety of platforms, such as Windows, Linux, and Mac OS.

## 4. CM CONFIGURATION

We assume that a developer wants to develop client and server applications using CM. In order to use CM, an application needs to import the CM library file and the Java database connectivity (JDBC) file, which are provided as Java archive (JAR) files. In addition, configuration files are also needed, which are provided with the CM JAR file. These configuration files must exist in the current working directory of the application under development.

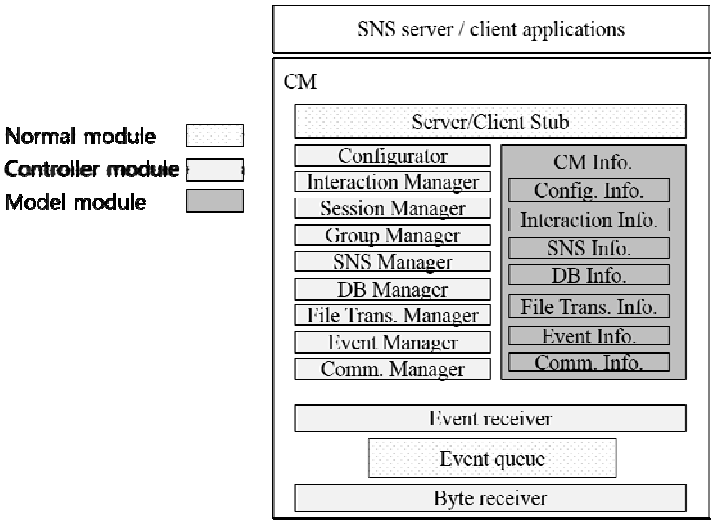


Figure 1. Communication middleware (CM) architecture

After the configuration files have been placed in the appropriate directory, the next step is to configure the default values within the files. A server application must set the CM server configuration files, the main one being the *cm-server.conf* file. This is a simple text file and can be edited by any text editor. Because most configuration fields have already been set to default values, the only thing the developer needs to do is to set the address of the default server. CM can configure multiple servers, consisting of the default server and additional servers; the default server is the one to which all clients must always connect. Thus, a server application using CM can be the default server or any of the additional servers; in the case of a single-server system, the server application is the default server. In this paper, we will assume a single server.

In the server configuration file, the developer can set other communication-related policies supported by the CM as well; the details are given below.

- *SYS_TYPE*: application type. A server application must set this field as "SERVER", and a client as "CLIENT" in the client-server model.
- *COMM_ARCH*: communication architecture. This field designates the communication architecture of an application using CM. Possible values are "CM_CS" for the client-server model and "CM_PS" for the hybrid model, the latter of which uses multicast communication in addition to the client-server model.
- *LOGIN_SCHEME*: user authentication policy. If the value is 0 (false), it means that the server will not authenticate a user when it receives a login request, and the server CM will always accept the request. If the value is 1 (true), the server will conduct the user authentication process.
- *SESSION_SCHEME*: multi-session policy. With this field, the developer can specify whether the application will use one session or multiple sessions. If the field value is 0, CM will not use multiple sessions but only a single default session; in this case, when a user logs in to a server, he or she automatically joins this session and its default group. If the value is 1, the server application can configure multiple sessions so that a user can select one of them to join.
- *DOWNLOAD_SCHEME*: transmission policy of SNS content. This field specifies how much SNS content a server will transmit to a client. If the value is 0, CM adopts a fixed amount of SNS content, according to the value of the *DOWNLOAD_NUM* field. If the value is 1, CM uses a dynamic downloading scheme [8].
- *DB_USE*: database (DB) usage flag. This field sets whether a server application uses CM's internal DB. If the value is 0, the application will not use the CM DB. If the value is 1, the application will use the CM DB, and the following additional DB information must be supplied: DB host, user name, password, port number, and DB name.
- *UDP_PORT*: default port number of the server application. Using CM, applications can send a message with a UDP connection. This field assigns the port number of the server application as the default UDP connection, which is open when the server CM starts.
- *FILE_PATH*: default path for file transfers. CM refers to this path when performing file transfers. If a file is requested, the server or client searches for the file in this file path; if a client receives a file, CM

stores the file in this file path. If a server receives a file, CM stores the file in a sub-directory of the default path; the sub-directory is set to the name of the client that sent the file.

- *SESSION_NUM*: number of sessions. The value must be greater than or equal to 1 because CM uses at least one session.
- *SESSION_FILE#*: name of session configuration file. "#" is an integer (starting with 1) that acts to differentiate among sessions. In a session configuration file, the developer sets group information for that session, such as group names and multicast addresses.
- *SESSION_NAME#*: session name. "#" is an integer (starting with 1) that acts to differentiate among sessions. Because a session name in CM is an identifier, a unique name must be assigned.

Likewise, a client application must set a CM client configuration file (*cm-client.conf*). Unlike the CM server configuration file, the CM client configuration file does not have many fields. The developer needs to specify system type, default server information, UDP port number, and file path information.

## 5. APPLICATION INTEGRATION

When the developer has finished setting the CM configuration files, he or she is now ready to develop client and server applications using CM. In order to initialize and start CM, both the server and the client applications need to declare an instance of the CM stub class and set a CM event handler object. Figure 2 shows sample code for a server application. Code for a client is nearly identical; the only difference is that it declares the instance of the CM client stub class instead of the server stub.

```
import kr.ac.konkuk.ccslab.cm.*;

public class CMServerApp {
	private CMServerStub m_serverStub;
	private CMServerEventHandler m_eventHandler;

	public CMServerApp()
	{
		m_serverStub = new CMServerStub();
		m_eventHandler = new CMServerEventHandler(m_serverStub);
	}

	public CMServerStub getServerStub()
	{
		return m_serverStub;
	}

	public CMServerEventHandler getServerEventHandler()
	{
		return m_eventHandler;
	}

	public static void main(String[] args) {
		CMServerApp server = new CMServerApp();
		CMServerStub cmStub = server.getServerStub();
		cmStub.setEventHandler(server.getServerEventHandler());
		cmStub.startCM();
		// start application routine
	}
}
```

Figure 2. Server application code

The registered event handler is called by CM whenever it receives a CM event so that an application can be notified of event reception. The developer should define an event handler class that includes an event process code. Figure 3 shows sample code for the server event handler. A client should define the event handler in the same manner.

```
import kr.ac.konkuk.ccslab.cm.*;

public class CMServerEventHandler implements CMEventHandler {
	private CMServerStub m_serverStub;

	public CMServerEventHandler(CMServerStub serverStub)
	{
		m_serverStub = serverStub;
	}

	@Override
	public void processEvent(CMEvent cme) {
		switch(cme.getType())
		{
		case CMInfo.CM_SESSION_EVENT:
			processSessionEvent(cme);
			break;
		// ... add other event types
		default:
			return;
		}
	}

	private void processSessionEvent(CMEvent cme)
	{
		// add the processing routine of session events
	}
}
```

Figure 3. Server event handler code

Once the CM is initialized and starts to run, an application can call CM APIs provided through the stub class.

## 6. PARTICIPATION IN CM NETWORK

A CM client must log in to the default server in order to interact with other CM nodes. For the login process, the CM client stub provides the *loginCM* method. This method takes two parameters: user name and password. When a client calls this method, the client CM sends a login request to the default server. When the server CM receives the login request, it authenticates the requesting user according to the CM login scheme specified in the CM server configuration file. If the server CM *LOGIN_SCHEME* value is 0, it does not authenticate users, instead accepting every login request; in this case, the server and client applications do not need to do anything further after the login request. If the server CM *LOGIN_SCHEME* value is 1, the server application is responsible for authenticating the requesting user with its own authentication policy. To this end, the server needs to capture the login request event in the event handler, authenticate the user, and notify the user of the result. A simple authentication technique is to use the CM DB manager, which provides an authentication method. To check whether the login request is successful, the client needs to capture the reply event. If the result field of the reply event is 1, the login request has successfully completed; otherwise, the login process has failed.

After the login process has completed, a client must join a session and a group to finish entering the CM network. The session join process is different according to whether the server has adopted a single session or multiple sessions in its configuration file. If there is a single session, the client CM automatically requests to join the session as soon as the login request finishes; thus, the client application need do nothing explicitly for joining a session. If there are multiple sessions available, the client needs to request session information, choose one session, and request to join that session through the CM client stub. When a client joins a session, it automatically proceeds to join the default group of that session.

## 7. CM COMMUNICATION SERVICES

For building an SNS application, CM provides various communication-related functionalities, including event management, user management, friend management, SNS content management, chat, and file transfer. In this section, we describe the roles of these services and how an application can use them.

### 7.1. Event Management

As CM handles every outgoing and incoming message as a type of CM event, an application needs to create an appropriate event in order to send a message. For simplicity, *CMDummyEvent* is one of the event classes supported by CM and has only one string field. This event is useful when the developer wishes to design a simple event that contains the semantic in a string variable. To support the more flexible format of a user-defined event, CM also provides the CM event class *CMUserEvent*. Using this event class, the developer can define an event field with a field data type, a field name, and its value. Normal data types such as int, long, float, double, char, String, and byte can be used in each event field. A created event field can be added to the instance of the *CMUserEvent* class. A *CMUserEvent* event that has its own event fields is identified by a string identifier, the name of which is also defined by the developer. The required methods to set and get the identifier and event fields are described in Table A1 of Appendix A.

The *setStringID* method is used to define the ID of a user-defined event. The developer should note that the data type of an ID is String rather than int. Although the pre-defined CM events use an ID of type int, the *CMUserEvent* class uses the String type because it offers more readability for a user-defined event. The ID of a user event can be retrieved by calling the *getStringID* method.

To set an event field in a user event, the *setEventField* method is used. This method requires three parameters: data type, field name, and field value. For the data type, CM provides six primitive types that are named as constant values, shown in Table 1.

The field data type and the field name parameters are required for identifying the different event fields in a user event. The last parameter is the field value. This value must always be given as String type, no matter which data type is used for the event field. In order to retrieve a field value, the *getEventField* method can be called. This method requires the field data type and the field name as parameters and returns the value of the corresponding event field. Because the return value is String type, the developer should transform it to the original data type if needed.

Table 1. Field Data Types of CMUserEvent Event

| Field data type | Matching Java data type |
|---|---|
| CMInfo.CM_INT | int |
| CMInfo.CM_LONG | long |
| CMInfo.CM_FLOAT | float |
| CMInfo.CM_DOUBLE | double |
| CMInfo.CM_CHAR | char |
| CMInfo.CM_STR | String |

The *setEventBytesField* method is used to set an event field that is an arbitrary byte stream. For example, bytes read from a file can be an event field. This method requires three parameters. The first parameter is the field name, identifying the event field of the byte stream. The second parameter is the number of bytes in the byte stream. The last parameter is a byte array that contains the bytestream value. The byte array can be retrieved by the *getEventBytesField* method. Requiring only the field name parameter, this method returns the bytestream value as a byte array.

The example in Figure 4 shows how a CM client creates a sample user event. The ID of the event is “testID”, and two event fields with different data types are set to the event. The client sends the created event to the default server.

```
...
CMUserEvent ue = new CMUserEvent();
int nField = 1;
String strField = "test string";
ue.setStringID("testID");
ue.setEventField(CMInfo.CM_INT, "intField", String.valueOf(nField));
ue.setEventField(CMInfo.CM_STR, "strField", strField);
m_clientStub.send(ue, "SERVER");
```

Figure 4. Example of CMUserEvent creation

A CM server or client application can send an event in any one of three transmission modes: one-to-one, one-to-many, or one-to-all transmission. In one-to-one transmission, there is only one receiver. In one-to-many mode, a sender can designate session members or group members as the recipients of the event. In

one-to-all mode, the event is sent to all logged-in users. Such transmissions are realized by three methods of the CM stub: the *send*, *cast* (or *multicast*), and *broadcast* methods.

To receive a user-defined event, a CM application can capture a user event of the *CMUserEvent* type, like other CM events. The example in Figure 5 shows how the default server captures the user event sent by the previous example and prints out the field values in the server event handler.

```
...
public void processEvent(CMEvent cme) {
        switch(cme.getType())
        {
        case CMInfo.CM_USER_EVENT:
                processUserEvent(cme);
                break;
        default:
                return;
        }
}

private void processUserEvent(CMEvent cme)
{
        CMUserEvent ue = (CMUserEvent) cme;
        System.out.println("event ID: "+ue.getStringID());
        nField = Integer.parseInt( ue.getEventField(CMInfo.CM_INT, "intField") );
        System.out.println("intField: "+ nField);
        strField = ue.getEventField(CMInfo.CM_STR, "strField");
        System.out.println("strField: " + strField);
}
```

Figure 5. Receiving a CMUserEvent event

### 7.2. User Management

The user management functions of CM include registration, deregistration, and user search. Because user profile information should be stored in a DB, the developer must set the server configuration file to use the CM DB (via *DB_USE* and other relevant fields) in order to use the user management support functions. User management APIs are provided by the CM client stub, as shown in Figure 6. A CM client can send a user management request to the default server, and the default server then processes the request using the CM DB. The default server sends the result of the request as a CM event that can be captured by the client event handler, as before.

```
void registerUser(String strName, String strPasswd)
void deregisterUser(String strName, String strPasswd)
void findRegisteredUser(String strName)
```

Figure 6. Methods for user management

A user can be registered to CM by the *registerUser* method of the CM client stub. If a CM client is connected to the default server, it can call this method. CM uses the registered user information to authenticate a user when that user logs in to the default server. The *registerUser* method requires only two parameters: user name (*strName*) and password (*strPasswd*). If the user name already exists in the CM DB, the registration fails; if the user name is uniquely specified, the registration succeeds. The success status of the registration request is assigned to the return code of the reply session event *REGISTER_USER_ACK*. If the request is successful, the reply event also contains the registration time at the server. The details of the *REGISTER_USER_ACK* event are shown in Table A2 of Appendix A.

A user can cancel his or her registration from CM by the *deregisterUser* method of the CM client stub. If a client is connected to the default server, it can call this method. When requested, CM removes the registered user information from the CM DB. Like the *registerUser* method, the *deregisterUser* method requires only two parameters: a user name (*strName*) and password (*strPasswd*). If the given user name with the correct password exists in the CM DB, the deregistration request is successful; otherwise, the request fails. The success status of the deregistration request is assigned to the return code of the reply session event *DEREGISTER_USER_ACK*, described in Table A3 of Appendix A.

A user can search for another user by the *findRegisteredUser* method of the CM client stub. If a client is connected to the default server, it can call this method. When requested, CM provides the basic profile of the target user, including such information as name and registration time. The *findRegisteredUser* method requires only the user name parameter. If the given user name exists in the CM DB, the search request is successful; otherwise, the request fails. The success status of the user search request is assigned to the return code of the reply session event *FIND_REGISTERED_USER_ACK*, described in Table A4 of Appendix A.

### 7.3. Friend Management

In an SNS application, it is common to support the management of friends for a user. CM makes it easy for a client to add, delete, and get friend information. In order to use this service, the developer needs to specify use of the CM DB, as in the user management case. Figure 7 shows a synopsis of the relevant methods of the CM client stub.

```
void addNewFriend(String strFriendName)
void removeFriend(String strFriendName)
void requestFriendsList()
void requestFriendRequestersList()
void requestBiFriendsList()
```

Figure 7. Methods for friend management

A client can call the *addNewFriend* method to add a new friend. A client can add a user as its friend only if the user name has already been registered to CM. If the friend is a registered user, the server adds it to the friend table of the CM DB as a friend of the requesting user; otherwise, the request fails. In either case, the server sends the reply event *ADD_NEW_FRIEND_ACK* with the result code to the requesting client to inform it of the request result. A client can delete a friend by calling the *removeFriend* method, and the result of the request, *REMOVE_FRIEND_ACK*, is sent to the requesting client. The event fields of both *ADD_NEW_FRIEND_ACK* and *REMOVE_FRIEND_ACK* are identical and are described in Table A5 of Appendix A.

Different SNS applications use the friend concept in different ways. In some applications, a user can add another user to his or her friend list without needing the agreement of the target user. In other applications, a user can add a friend only if the other user accepts the friend request. CM supports these different policies of friend management by providing methods that request different user lists. The *requestFriendsList* method requests the list of users whom the requesting user has added as friends, regardless of their acceptance by the others. The *requestFriendRequestersList* method requests the list of users who have added the requesting user as a friend but whom the requesting user has not as yet added as friends. The *requestBiFriendsList* method requests the list of users who have added the requesting user as a friend and whom the requesting user has added as friends. To illustrate, three different friend relationships can be represented as a directed graph, shown in Figure 8.

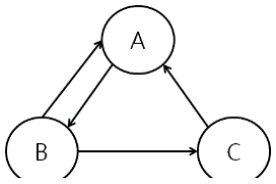


Figure 8. Different friend relationships

A node represents a user, and a directed edge is a friend relationship. In the figure, there are three users, *A*, *B*, and *C*. The existence of a directed edge from user *A* to user *B* means that *A* has added *B* as its friend. If there are edges in both directions between two nodes, the corresponding users have added each other, becoming bilateral friends. If users *A*, *B*, and *C* now call the above three methods, the result will be as shown in Table 2.

Table 2. Results of Calling Three Friends-related Methods

| Method name | A result | B result | C result |
|---|---|---|---|
| *requestFriendsList* | B | A,C | A |
| *requestFriendRequestersList* | C | N/A | B |
| *requestBiFriendsList* | B | A | N/A |

When the default server receives the request for friends, requesters, or bilateral friends from a client, it sends the corresponding user list as the *RESPONSE_FRIEND_LIST*, *RESPONSE_FRIEND_REQUESTER_LIST*, or *RESPONSE_BI_FRIEND_LIST* event to the requesting client. The three events have the same event fields, which are described in Table A6 of Appendix A. One of the event fields is the friend list, but the meaning of the list differs according to the event ID. The friend list contains a maximum of 50 user names. If the total number exceeds 50, the server will send the event more than once.

### 7.4. SNS Content Management

Using the SNS content service of CM, a client application can request to upload and download SNS content. For uploading or downloading content, a client can call the *requestSNSContentUpload* or *requestSNSContent* method, respectively, in the CM client stub. Figure 9 shows a synopsis of these methods. For persistence of SNS content, the CM default server stores the uploaded content in an SNS content table of the CM DB. Therefore, the CM server must be set to use the CM DB. To request content upload or download, a client must log in to the default server.

```
void requestSNSContentUpload(String user, String message, int
        nNumAttachedFiles, int nReplyOf, int nLevelOfDisclosure,
        ArrayList<String> filePathList)
void requestSNSContent(String strUser, String strWriter, int nOffset)
```

Figure 9. Methods for content upload and download

A client can call the *requestSNSContentUpload* method to upload a message to the default server. This method requires six parameters. The first parameter, *user*, is the name of the user who uploads a message. The second parameter, *message*, is a text message. The third parameter, *nNumAttachedFiles*, is the number of attached files in this message; this parameter value must be the same as the number of elements in the file path list specified as the last parameter. The fourth parameter, *nReplyOf*, indicates the ID number (greater than 0) of the content to which this message replies; a value of 0 indicates that the uploaded content is original rather than a reply. The fifth parameter, *nLevelOfDisclosure*, specifies the level of disclosure (LoD) of the uploaded content. CM allows four levels of disclosure of content: LoD 0 opens the uploaded content to public viewing, LoD 1 allows only those users who have added the uploading user as a friend to access the uploaded content, LoD 2 allows only bilateral friends of the uploading user to access the uploaded content, and LoD 3 makes the uploaded content private. The last parameter, *filePathList*, is the list of attached files. Path names of attached files should be given as type ArrayList, and the number of array elements must be the same as that specified as the value of the *nNumAttachedFiles* parameter.

If the server receives a content upload request, it stores the requested message with the user name, the index of the content, the upload time, the number of attachments, the reply ID, and the level of disclosure. If the content has files attached, the client transfers them separately to the server. After the upload task has completed, the server sends the *CONTENT_UPLOAD_RESPONSE* event to the requesting client so that the client can check the result of the request. The event fields of the *CONTENT_UPLOAD_RESPONSE* event are detailed in Table A7 of Appendix A.

A client can request to download content by calling the *requestSNSContent* method. The first parameter of this method, *strUser*, is the name of the requesting user. The second parameter, *strWriter*, specifies a user who has uploaded content. For this parameter, the client can designate either a specific writer name or a friend group. If the parameter value is a specific user name, the client downloads only content that has been uploaded by the specified name and that is accessible by the requester. If the parameter value is "CM_MY_FRIEND", the client downloads content that was uploaded by the requester's friends. If the parameter value is "CM_BI_FRIEND", the client downloads content that was uploaded by the requester's bilateral friends. If the parameter value is an empty string (""), the client does not specify a writer name, and it downloads all content that the requester is eligible to access. The last parameter, *nOffset*, is an offset from

the beginning of the requested content list, specifying that the client wants to download some number of SNS messages starting from the *nOffset*-th most recent message. The *nOffset* value must be greater than or equal to 0.

When the server receives the download request, it first determines how many SNS messages will be sent. The number of messages is decided by the *DOWNLOAD_SCHEME* field of the server configuration file. If this field is set to 0, the server uses the fixed maximum number of messages per request, according to the value of the *DOWNLOAD_NUM* field of the configuration file. If *DOWNLOAD_SCHEME* is set to 1, the server uses our dynamic downloading scheme, which determines the number of downloaded messages according to the round-trip delay between the server and the requesting client. Each SNS message is then sent to the client as a *CONTENT_DOWNLOAD* event, which can be captured in the client event handler. The fields of the *CONTENT_DOWNLOAD* event are described in Table A8 of Appendix A. In most cases, the server sends multiple *CONTENT_DOWNLOAD* events, corresponding to the number of SNS messages, and then it sends the *CONTENT_DOWNLOAD_END* event to signal the end of the current download. This event contains a field that gives the number of downloaded messages. A client can send another download request by updating the offset parameter with the number of previously downloaded messages. The detailed event fields of the *CONTENT_DOWNLOAD_END* event are described in Table A9 of Appendix A. If the content has attached files, the client stores them separately in the default directory that is specified in the *FILE_PATH* field of the client configuration file. The client should use the directory information from the *CMFileTransferInfo* classes in order to access the downloaded files because the *CONTENT_DOWNLOAD* event includes only the attached file names.

### 7.5. Chat Management

Another way clients frequently interact with other clients is via chat. A client can send a chat message simply by calling the *chat* method of the CM client stub. This method takes two string parameters: a target and a text message. Using the target parameter of the *chat* method, we can easily control the range of recipients. The top level of the range is "/b", which broadcasts the chat event to all logged-in users. The "/s" value is used if a sender wants to chat with only those users who are in the same current session. The "/g" value limits the recipients to the current group members. In the last case there is only one receiver, whose name is designated after the "/" character as the first parameter of the *chat* method.

A CM application can receive a chatting event by capturing a pre-defined CM event in the event handler, in the same manner as other events. There are two types of CM chat events. One is the *SESSION_TALK* event of the *CMSessionEvent* class; a client can receive this event if it is logged in at least to the default server. The other event is the *USER_TALK* event of the *CMInterestEvent* class; a client can receive this event only if it joins a group. Tables A10 and A11 of Appendix A, respectively, describe the detailed field information for these events.

### 7.6. File Transfer Management

CM applications that connect directly to each other can exchange a file using the *CMStub* class, which is the parent class of the *CMClientStub* and *CMServerStub* classes. For example, in client-server architecture, a client can push or pull a file to or from a server by calling the *pushFile* or *requestFile* methods. In order to use the file transfer service, a CM application must set a directory to be the file repository. When CM is initialized by an application, the default directory is configured in the configuration file (by the *FILE_PATH* field). If a file is requested, the server or client searches for the file in this file path; if a client receives a file, CM stores the file in this file path. If a server receives a file, CM stores the file in a sub-directory of the default path; the sub-directory is set to the name of the client that sent the file. An application can change the default file path by the *setFilePath* method.

In pull mode, a CM application requests a file from another remote CM application. For example, a CM client can request that a CM server send a file. A file is requested by the *requestFile* method. The *requestFile* method requires two parameters: the requested file name and the name of the file owner. In push mode, on the other hand, a CM application can send a file to another remote CM application. A file is pushed by the *pushFile* method. Like the *requestFile* method, the *pushFile* method requires two parameters: the path name of the file to be sent and the name of the receiver.

## 8. CONCLUSION

In this paper, we have introduced our communication middleware (CM), with which a developer can easily implement common communication services for SNS applications in a client-server architecture. The CM configuration enables an SNS server to specify a policy for user authentication, session and group organization for user interactions, and a policy for SNS content download. Using the CM APIs, it is easy for

an SNS client to compose and send a user-defined event in different transmission modes, request user registration and authentication, add and remove friend lists according to different concepts of friend relationships, upload and download SNS content with different levels of disclosure, send a chat message to different ranges of recipient groups, and push or pull a file to or from a server. As the CM communication services are fundamental components for various interactions of distributed nodes, these services can also be applied in the development of other distributed applications.

For future work, we plan to add more communication services, such as cloud storage and location detection, in order to enrich the range of CM support, particularly for mobile SNSs. We are also researching an adaptive content transmission and prefetching scheme in order to reduce delays in accessing high-volume and high-quality social content under poor network conditions.

## ACKNOWLEDGEMENTS

This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2015-H8501-15-1004) supervised by the IITP (Institute for Information & communications Technology Promotion).

## REFERENCES

[1] D. Brooker, *et al.*, "Middleware for Social Networking on Mobile Devices", *in 21st Australian Software Engineering Conference*, April 6-9, 2010, pp. 202-211.
[2] A. Pietilainen, *et al.*, "MobiClique: Middleware for Mobile Social Networking", *in 2nd ACM Workshop on Online Social Networks*, August 17, 2009, pp. 49-54.
[3] A. Gupta, *et al.*, "MobiSoC: A Middleware for Mobile Social Computing Applications", *Journal of Mobile Networks and Applications*, vol. 14, pp. 35-52, February 2009.
[4] S. Mokhtar, *et al.*, "A Middleware Service for Pervasive Social Networking", *in International Workshop on Middleware for Pervasive Mobile and Embedded Computing*, December 2009, Article No. 2.
[5] B. Karki, *et al.*, "Social Networking on Mobile Environment", *in ACM/IFIP/USENIX 9th International Middleware Conference*, December 1-5, 2008, pp. 93-94.
[6] J. Porras, *et al.*, "Peer-to-Peer Communication Approach for A Mobile Environment", *in 37th Annual Hawaii International Conference on System Sciences (HICSS)*, 2004.
[7] M. Lim, "Improving Architecture of Communication Middleware for Social Networking Services", *Technical Report*, CCSLab, Konkuk University, 2014.
[8] M. Lim, "Adaptation of Content Transmission for Social Network Systems", *International Journal of Information Processing and Management (IJIPM)*, vol. 4, pp.60-67, September 2013.

## BIOGRAPHIES OF AUTHORS

Yunjin Lee is an Associate professor of in the Division of Digital Media at Ajou University. She received her BS degree in 1999 and her PhD degree in 2005, all in Computer Science and Engineering from POSTECH in Korea. Her research interests include nonphotorealistic rendering, 3D mesh processing, and data compression.

Mingyu Lim is an Associate Professor at the Department of Internet and Multimedia Engineering, Konkuk University, Korea from the year 2009. Before joining Konkuk University, he was a senior researcher at MIRALab, University of Geneva, being involved in various research activities on networked virtual environments and ubiquitous computing systems. He received his PhD in Computer Science in February 2006 at ICU, Korea. His major research field is supporting scalability in networked virtual environments. His current research activities are focused on communication middleware, event transmissions, and content distribution in distributed systems and Internet of Things.

Yangchan Moon is a Ph.D. student at the Department of Internet and Multimedia Engineering, Konkuk University, Korea from the year 2015. He received his BS degree in 2012 and his Master degree in 2015, all in Internet and Multimedia Engineering from Konkuk University in Korea. His research interests include distributed systems, communication middleware, and content sharing and prefetching systems.

## Appendix A. CM Events for Notification to a Client

Table A1. Methods of CMUserEvent Class.

| Event type | CMInfo.CM_USER_EVENT |
|---|---|
| Methods | Usage |
| void setStringID(String id) | Set a String ID of this event |
| void setEventField(int type, String fName, String fValue) | add an event field |
| void setEventBytesField(String fName, intbyteNum, byte[] bytes) | add an event field which is a byte array |
| String getStringID() | Get a String ID of this event |
| String getEventField(int type, String fName) | Get the value of an event field |
| byte[] getEventBytesField(String fName) | Get the byte array of an event field |

Table A2. REGISTER_USER_ACK Event.

| Event Type | | CMInfo.CM_SESSION_EVENT | |
|---|---|---|---|
| Event ID | | CMSessionEvent.REGISTER_USER_ACK | |
| Event field | Field Data type | Field definition | Get method |
| Return code | int | Result code of the request<br>1: succeeded<br>0: failed | getReturnCode() |
| User name | String | Requester user name | getUserName() |
| Creation time | String | Time to create the user at DB | getCreationTime() |

Table A3. DEREGISTER_USER_ACK Event.

| Event Type | | CMInfo.CM_SESSION_EVENT | |
|---|---|---|---|
| Event ID | | CMSessionEvent.DEREGISTER_USER_ACK | |
| Event field | Field Data type | Field definition | Get method |
| Return code | Int | Result code of the request<br>1: succeeded<br>0: failed | getReturnCode() |
| User name | String | Requester user name | getUserName() |

Table A4. FIND_REGISTERED_USER_ACK Event.

| Event Type | | CMInfo.CM_SESSION_EVENT | |
|---|---|---|---|
| Event ID | | CMSessionEvent.FIND_REGISTERED_USER_ACK | |
| Event field | Field Data type | Field definition | Get method |
| Return code | Int | Result code of the request<br>1: succeeded<br>0: failed | getReturnCode() |
| User name | String | Requested user name | getUserName() |
| Creation time | String | Time to create the user at DB | getCreationTime() |

Table A5. ADD_NEW_FRIEND_ACK and REMOVE_FRIEND_ACK Events.

| Event Type | | CMInfo.CM_SNS_EVENT | |
|---|---|---|---|
| Event ID | | CMSNSEvent.ADD_NEW_FRIEND_ACK<br>CMSNSEvent.REMOVE_FRIEND_ACK | |
| Event field | Field Data type | Field definition | Get method |
| Return code | int | Result code of the request<br>1: succeeded; 0: failed | getReturnCode() |
| User name | String | The name of a requesting user | getUserName() |
| Friend name | String | The name of a friend to add or remove | getFriendName() |

Table A6. Response Events of the Three List Requests.

| Event Type / Event ID | CMInfo.CM_SNS_EVENT<br>CMSNSEvent.RESPONSE_FRIEND_LIST<br>CMSNSEvent.RESPONSE_FRIEND_REQUESTER_LIST<br>CMSNSEvent.RESPONSE_BI_FRIEND_LIST | | |
|---|---|---|---|
| Event field | Field Data type | Field definition | Get method |
| User name | String | The name of a requesting user | getUserName() |
| Total friend number | int | Total number of requested users | getTotalNumFriends() |
| Friend number | int | Number of requested users in this event | getNumFriends() |
| Friend list | ArrayList<String> | List of requested user names | getFriendList() |

Table A7. CONTENT_UPLOAD_RESPONSE Event.

| Event Type / Event ID | CMInfo.CM_SNS_EVENT<br>CMSNSEvent.CONTENT_UPLOAD_RESPONSE | | |
|---|---|---|---|
| Event field | Field Data type | Field definition | Get method |
| Return code | int | Result code of the request<br>1: succeeded; 0: failed | getReturnCode() |
| Content ID | int | An index of the uploaded content in a content table | getContentID() |
| Date and time | String | Date and time of the upload | getDate() |
| User name | String | Requesting user name | getUserName() |

Table A8. DOWNLOAD_CONTENT Event.

| Event Type / Event ID | CMInfo.CM_SNS_EVENT<br>CMSNSEvent.CONTENT_DOWNLOAD | | |
|---|---|---|---|
| Event field | Field Data type | Field definition | Get method |
| User name | String | Requester name | getUserName() |
| Offset | int | Requested content offset | getContentOffset() |
| Content ID | int | Content ID | getContentID() |
| Date and time | String | Written date and time of the content | getDate() |
| Writer name | String | Writer name of the content | getWriterName() |
| Text message | String | Text message of the content | getMessage() |
| No.of attachments | int | Number of attached files | getNumAttachedFiles() |
| Reply ID | int | Content ID to which this message replies (0 for no reply) | getReplyOf() |
| Level of disclosure | int | Level of disclosure of the message<br>0: open to public<br>1: open only to friends<br>2: open only to bi-friends<br>3: private | getLevelOfDisclosure() |
| File name list | ArrayList<String> | The list of attached file name | getFileNameList() |

Table A9. DOWNLOAD_CONTENT_END Event.

| Event Type / Event ID | CMInfo.CM_SNS_EVENT<br>CMSNSEvent.CONTENT_DOWNLOAD_END | | |
|---|---|---|---|
| Event field | Field Data type | Field definition | Get method |
| User name | String | Requester name | getUserName() |
| Offset | int | Requested content offset | getContentOffset() |
| Content ID | int | Content ID | getContentID() |
| Download number | int | Number of downloaded SNS messages | getNumContents() |

Table A10. SESSION_TALK Event.

| Event Type / Event ID | CMInfo.CM_SESSION_EVENT<br>CMSessionEvent.SESSION_TALK | | |
|---|---|---|---|
| Event field | Field Data type | Field definition | Get method |
| User name | String | Name of the sending user | getUserName() |
| Text message | String | A chatting message | getTalk() |
| Session name | String | A current session of the sending user | getHandlerSession() |

Table A11. USER_TALK Event.

| Event Type / Event ID | CMInfo.CM_INTEREST_EVENT<br>CMInterestEvent.USER_TALK | | |
|---|---|---|---|
| Event field | Field Data type | Field definition | Get method |
| User name | String | Name of the sending user | getUserName() |
| Text message | String | A chatting message | getTalk() |
| Session name | String | A current session of the sending user | getHandlerSession() |
| Group name | String | A current group of the sending user | getHandlerGroup() |